MPI on BlueGene/L: Designing an Efficient General Purpose Messaging Solution for a Large Cellular System

George Almási[1], Charles Archer[2], José G. Castaños[1], Manish Gupta[1], Xavier Martorell[1], José E. Moreira[1], William Gropp[3], Silvius Rus[4], and Brian Toonen[3]

[1] IBM T. J. Watson Research Center, Yorktown Heights NY 10598-0218
{gheorghe,castanos,jmoreira,mgupta,xavim}@us.ibm.com
[2] IBM Systems Group, Rochester MN 55901
archerc@us.ibm.com
[3] Argonne National Laboratory, Argonne IL 60439
{gropp,toonen}@mcs.anl.gov
[4] Texas A&M University, College Station TX 77840
rus@tamu.edu

Abstract. The BlueGene/L computer uses system-on-a-chip integration and a highly scalable 65,536-node cellular architecture to deliver 360 Tflops of peak computing power. Efficient operation of the machine requires a fast, scalable, and standards compliant MPI library. In this paper, we discuss our efforts to port the MPICH2 library to BlueGene/L.

1 Introduction

BlueGene/L [2] is a 65,536-compute node massively parallel system being developed by IBM in partnership with Lawrence Livermore National Laboratory. Through the use of system-on-a-chip integration [3], coupled with a highly scalable cellular architecture, BlueGene/L will deliver 360 Tflops of peak computing power.

In this paper we present and analyze the software design for a fast, scalable, and standards compliant MPI communication library, based on MPICH2 [1], for the BlueGene/L machine. MPICH2 is an all-new implementation of MPI that is intended to support both MPI-1 and MPI-2. The MPICH2 design features optimized MPI datatypes, optimized remote memory access (RMA), high scalability, usability, and robustness.

The rest of this paper is organized as follows. Section 2 presents a brief description of the BlueGene/L computer. Section 3 discusses its system software. Section 4 gives a high level architectural overview of the communication library, and Section 5 discusses the design choices we are facing during implementation. Section 6 describes the methodology we employed to measure performance and presents preliminary results. We conclude with Section 7.

2 BlueGene/L hardware overview

The basic building block of BlueGene/L is a custom chip that integrates processors, memory, and communications logic. A chip contains two 32-bit embedded PowerPC

440 cores with custom *dual* floating-point units that operate on two-element vectors. The theoretical peak performance of the chip is 5.6 Gflops at the target clock speed of 700 MHz. An external double data rate (DDR) memory system completes a BlueGene/L node. The complete BlueGene/L machine consists of 65,536 compute nodes and 1,024 I/O nodes. I/O nodes are connected to a Gigabit Ethernet network and serve as control and file concentrators for the compute nodes.

The compute nodes of BlueGene/L are organized into a partitionable $64 \times 32 \times 32$ three-dimensional torus network. Each compute node contains six bi-directional torus links for direct connection with nearest neighbors. The network hardware guarantees reliable and deadlock-free, but unordered, delivery of variable length (up to 256 bytes) packets, using a minimal adaptive routing algorithm. It also provides simple broadcast functionality by depositing packets along a route. At 1.4 Gb/s per direction, the bisection bandwidth of the system is 360 GB/s per direction. The I/O nodes are not connected to the torus network.

All compute and I/O nodes of BlueGene/L are interconnect by a tree network. The tree network supports fast point-to-point, broadcast, and reduction operations on packets, with a hardware latency of approximately 2 microseconds for a 64k-node system. An ALU in the network can combine incoming packets using bitwise and integer operations, forwarding a resulting packet along the tree. Floating-point reductions can be performed in two phases (one for the exponent and another one for the mantissa) or in one phase by converting the floating-point number to an extended 2048-bit representation.

3 BlueGene/L system software overview

The smallest unit independently controlled by software is called a processing set (or pset) and consists of 64 compute nodes and an associated I/O node. Components of a pset communicate through the tree network. File I/O and control operations are performed by the I/O node of the pset through the Ethernet network.

The I/O nodes run a Linux kernel with custom drivers for the Ethernet and tree devices. The main function of I/O nodes is to run a program called **ciod** (control and I/O daemon) that implements system management services and supports file I/O operations by the compute nodes.

The control software running on compute nodes is a minimalist POSIX compliant *compute node kernel* (CNK) that provides a simple, flat, fixed-size address space for a single user process. The CNK plays a role similar to PUMA [13] in the ASCI Red machine. The torus network is mapped directly into user space and the tree network is partitioned between the kernel and the user.

The system management software provides a range of services for the machine, including machine initialization and booting, system monitoring, job launch and termination, and job scheduling. System management is provided by external service nodes that act upon the I/O and compute nodes, both directly and through the **ciod** daemons. The partitioning of the system into compute, I/O, and service nodes leads to a hierarchical system management software.

4 Communication software architecture

The BlueGene/L communication software architecture is divided into three layers. At the bottom is the *packet layer*, a thin software library that allows access to network hardware. At the top is the MPI library. The *message layer* glues the packet layer and MPI together.

Packet layer. The torus/tree packet layer is a thin layer of software designed to abstract and simplify access to hardware. It abstracts hardware FIFOs into torus and tree *devices* and presents an API consisting of essentially three functions: initialization, packet send and packet receive. The packet layer provides a *mechanism* to use the network hardware but does not impose any *policies* on how to use it.

Some restrictions imposed by hardware are not abstracted at packet level for performance reasons. For example, the length of a torus packet must be a multiple of 32 bytes, and can be no more than 256 bytes. Tree packets have exactly 256 bytes. Packets sent and received by the packet layer have to be aligned to a 16-byte address boundary, to enable the efficient use of 128-bit loads and stores to the network hardware through the dual floating-point units.

All packet layer send and receive operations are non-blocking, leaving it up to the higher layers to implement synchronous, blocking and/or interrupt driven communication models. In its current implementation the packet layer is stateless.

Message layer. The message layer is an active message system [7, 12, 14, 15], built on top of the packet layer, that allows the transmission of arbitrary buffers among compute nodes. Its architecture is shown by Figure 1.

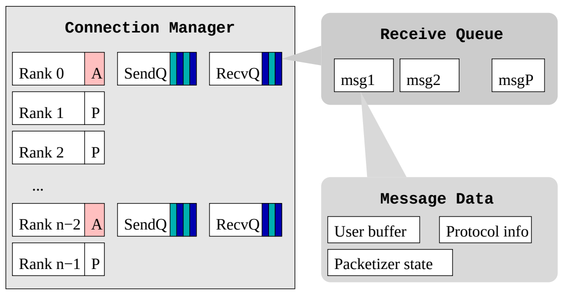

Fig. 1. The message layer architecture

The connection manager controls the overall progress of the system and contains a list of virtual connections to other nodes. Each virtual connection is responsible for communicating with one peer. The connection has a send queue and a receive queue. Outgoing messages are always sent in order. Incoming packets, however, can arrive out of order. The message layer has to determine which message a packet belongs to. Thus, each packet has to carry a message identifier.

Message buffers are used for sending and receiving packets belonging to the same message. A message buffer contains the state of the message (in progress, complete, etc). It also has an associated region of user memory, and a *packetizer/unpacketizer* that is able to generate packets or to place incoming packets into memory. Message buffers also handle the message protocol (*i.e.*, what packets to send when).

Packetizers and unpacketizers drive the packet layer. Packetizers build and send packets out of message buffers. Unpacketizers rebuild messages from the component packets. Packetizers also handle the alignment and packet size limitations imposed by the network hardware.

The three main functions implemented by the message layer API are `Init`, `advance` and `postsend`: `Init` initializes the message layer; `advance` is called to ensure that the message layer makes progress (*i.e.*, sends the packets it has to send, checks the torus hardware for incoming packets and processes them accordingly); `postsend` allows a message to be submitted into the send queue.

Just like packet layer functions, message layer functions are non-blocking and designed to be used in either polling mode, or driven by hardware interrupts. Completion of a send, and the beginning and end of a receive are all signaled through *callbacks*. Thus, when a message is sent and is ready to be taken off the send queue the `senddone` function is invoked. When a new message starts arriving, the `recvnew` callback is invoked. At the end of reception `recvdone` is invoked.

MPICH2. MPICH2, currently under development at Argonne National Laboratory, is an MPI implementation designed from the ground up for scalability to hundreds of thousands of processors. Figure 2 shows the roadmap of developing an MPI library for BlueGene/L. MPICH2 has a modular structure, and therefore the BlueGene/L port consists of a number of plug-in modules, leaving the code structure of MPICH2 intact.

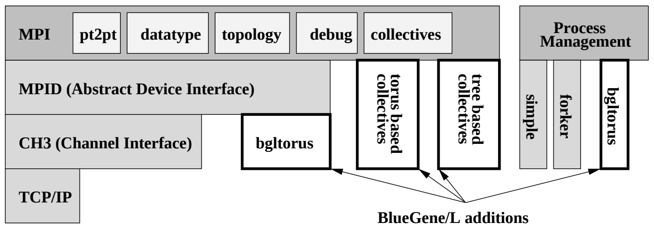

Fig. 2. The BlueGene/L MPI roadmap.

The most important addition of the BlueGene/L port is an implementation of ADI3, the MPICH2 Abstract Device Interface [9]. A thin layer of code transforms (for example) MPI `Request` objects and `MPI_Send` function calls into sequences of message layer `postsend` function calls and various message layer callbacks.

Another part of the BlueGene/L port is related to the process management primitives. In MPICH2, process management is split into two parts: a process management interface (PMI), called from within the MPI library, and a set of process managers (PM) which are responsible for starting up and terminating down MPI jobs and implementing the PMI functions. The BlueGene/L process manager makes full use of its hierarchical system management software to start up and shut down MPI jobs, dealing with the scalability problem inherent in starting up, synchronizing, and killing 65,536 MPI processes.

MPICH2 has default implementations for all MPI collectives and becomes functional the moment point-to-point primitives are implemented. The default implementations are oblivious of the underlying physical topology of the torus and tree networks. Optimized collective operations can be implemented for communicators whose physical layouts conform to certain properties. Building optimized collectives for MPICH2 involves several steps. First, the process manager interface needs to be expanded to allow the calculation of the torus and tree layouts of particular communicators. Next, a list of optimized collectives, for particular combinations of communicator layouts and message types, needs to be implemented. The best implementation of a particular MPI collective will be selected at run-time, based on the type of communicator involved (as calculated using the process manager interface).

– The torus hardware can be used to efficiently implement broadcasts on contiguous 1-, 2-, and 3-dimensional meshes, using the feature of the torus that allows depositing a packet on every node it traverses. Collectives best suited for this implementation include `Bcast`, `Allgather`, `Alltoall`, and `Barrier`.
– The tree hardware can be used for almost every collective that is executed on the `MPI_COMM_WORLD` communicator, including reduction operations.
– Non-`MPI_COMM_WORLD` collectives can also be implemented using the tree, but care must be taken to ensure deadlock free operation. The tree network guarantees deadlock-free simultaneous delivery of two virtual channels. One of these channels is used for control and file I/O purposes; the other is available for use by collectives.

5 Design decisions in the message layer

The design of the message layer was influenced by specific BlueGene/L hardware features, such as network reliability, packetization and alignment restrictions, out-of-order arrival of torus packets, and the existence of non-coherent processors in a chip. These hardware features, together with the requirements for a low overhead scalable solution, led us to design decisions that deserve closer examination.

The impact of hardware reliability. The BlueGene/L network hardware is completely reliable. Once a packet is injected into the network, hardware guarantees its arrival at the destination. The BlueGene/L message layer does not implement a packet recovery protocol, allowing for better scaling and a large reduction of software overhead.

Packetizing and alignment. The packet layer requires data to be sent in (up to) 256-byte chunks aligned at 16-byte boundaries. This forces the message layer to either optimize the alignment of arbitrary buffers or to copy memory to/from aligned data buffers.

Figure 3 illustrates the principle of optimizing the alignment of long buffers. The buffer is carved up into aligned chunks where possible. The two non-aligned chunks at the beginning and at the end of the buffer are copied and sent together. This strategy is not always applicable, because the alignment *phase* (*i.e.*, the offset from the closest aligned address) of the sending and receiving buffers may differ. MPI has no control over the allocation of user buffers. In such cases at least one of the participating peers, preferably the sender, has to adjust alignment by performing memory to memory copies. For rendezvous messages the receiver can send back the *desired alignment phase* with the first acknowledgment packet. We note that the alignment problem only affects zero copy message sending strategies, since a memory copy can absorb the cost of re-alignment.

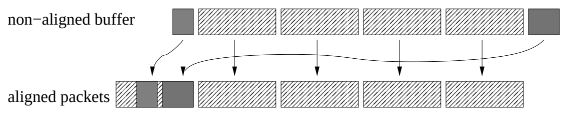

Fig. 3. Packetizing non-aligned data.

Out-of-order packets. The routing algorithm of the torus network allows packets from the same sender to arrive at the receiver out of order. The task of re-ordering packets falls to the message layer.

Packet order anomalies affect the message layer in one of two ways. The simpler case occurs when packets belonging to the same message are received out of order. This affects the way in which packets are re-assembled into messages, and the way in which MPI matching is done at the receiver (since typically MPI matching information is in the *first* packet of a message).

Packet order reversal can also occur to packets belonging to different messages. To prevent the mixing of packets belonging to different messages, each packet has to carry a message identifier. To comply with MPI semantics, the receiver is responsible to present incoming messages to MPI in strictly increasing order of the message identifier.

Cache coherence and processor use policy. Each BlueGene/L compute node incorporates two non-cache-coherent PowerPC 440 cores sharing the main memory and devices. Several modes of operation for these two cores have been proposed.

– *Heater mode* puts the second core into an idle loop. It is easy to implement because it sidesteps all issues of cache coherency and resource sharing.
– *Virtual node mode* executes a different application process in each core. Virtual node mode doubles the processing power available to the user at the cost of halving all other resources per process. It is well suited for computation-intensive jobs that require little in the way of memory or communication. We are still in the beginning of our research on virtual node mode and do not discuss it further in this paper.
– *Communication coprocessor mode* is considered the default mode of operation. It assigns one processor to computation and another to communication, effectively overlapping them by freeing the compute processor from communication tasks.

The main obstacle to implementing communication coprocessor mode efficiently is the lack of cache coherence between processors. A naive solution is to set up a non-cached shared memory area and implement a virtual torus device in that area. The computation processor communicates only with the virtual torus device. The communication processor simply moves data between the real device and the virtual device.

The naive implementation of coprocessor mode still requires the compute processor to packetize and copy data into the shared memory area. However, reads and writes to/from the shared memory area can be done about four times faster than to/from the network devices, reducing the load on the compute processor by the same amount.

For better performance, we want to develop a mechanism in which the communication processor moves data to and from the application memory directly. Before sending an MPI message, the compute processor has to insure that the application buffer has been flushed to main memory. The communication processor can then move that data directly to the hardware torus. When receiving a message, the compute processor has to invalidate the cache lines associated with its receive buffer before allowing the communication processor to fill it in with incoming data.

Scaling issues and virtual connections. In MPICH2, point to point communication is executed over virtual connections between pairs of nodes. Because the network hardware guarantees packet delivery, virtual connections in BlueGene/L do not have to execute a per-connection wake-up protocol when the job starts. Thus startup time on the BlueGene/L machine will be constant for any number of participating nodes.

Another factor limiting scalability is the amount of memory needed by an MPI process to maintain state for each virtual connection. The current design of the message layer uses only about 50 bytes of data for every virtual connection for the torus coordinates of the peer, pointer sets for the send and receive queues, and state information. Even so, 65,536 virtual connections add up to 3 MBytes of main memory per node, or more than 1% of the available total (256 MBytes), just to maintain the connection table.

Transmitting non-contiguous data. The MPICH2 abstract device interface allows non-contiguous data buffers to percolate down to message layer level, affording us the opportunity to optimize the marshalling and unmarshalling of these data types at the lowest (packet) level. Our current strategy centers on `iovec` data structures generated by utility functions in the ADI [9] layer.

Communication protocol in the message layer. Early in the design we made the decision to implement the communication protocol in the message layer for performance reasons. Integration with MPICH2 is somewhat harder, forcing us to implement an abstract device interface (ADI3) port instead of using the easier, but less flexible, channel interface [9]. In our view, the additional flexibility gained by this decision is well worth the effort. The protocol design is crucial because it is influenced by virtually every aspect of the BlueGene/L system: the reliability and out of order nature of the network, scalability issues, and latency and bandwidth requirements.

– Because of the reliable nature of the network no acknowledgments are needed. A simple “fire and forget” eager protocol is a viable proposition. Any packet out the send FIFO can be considered safely received by the other end.

– A special case of the eager protocol is represented by single-packet messages, which should be handled with a minimum of overhead to achieve good latency.
– The main limitation of the eager protocol is the inability of the receiver to control incoming traffic. For high volume messages, the rendezvous protocol is called for, possibly the optimistic form implemented in Portals [5].
– The message protocol is also influenced by out-of-order arrival of packets. The first packet of any message contains information not repeated in other packets, such as the message length and MPI matching information. If this packet is delayed on the network, the receiver is unable to handle the subsequent packets, and has to allocate temporary buffers or discard the packets, with obvious undesirable consequences. This problem does not affect the rendezvous protocol because the first packet is always explicitly acknowledged and thus cannot arrive out of order.
– A solution to the out-of-order problem for mid-size messages involves a variation on the rendezvous protocol that replicates the MPI matching information in the first few packets belonging to a message, and requires the receiver to acknowledge the first packet it receives. The number of packets that have to carry extra information is determined by the average roundtrip latency of the torus network. The sender will not have to stop and wait for an acknowledgment if it is received before the allotment of special packets carrying extra information has been exhausted.

6 Simulation framework and measurements

This section illustrates the measurement methodology we are using to drive our design decisions, and how we are planning to optimize the implementation of the MPI port. The numbers presented here are current as of April 2003, and were measured with the first version of the BlueGene/L port of MPICH2 that was able to run in the `BGLsim` multichip simulation environment [6]. `BGLsim` is equipped with an implementation of HPM [8, 10] which allows us to measure the number of instructions executed by regions of instrumented code. We measure the software overhead in the MPICH2 port and in the message layer. The workloads for our experiments consisted of the NAS Parallel Benchmarks [4, 11], running on 8 or 9 processors, depending on the benchmark.

Figure 4 shows a simplified call graph for sending a blocking MPI message, with the functions of interest to us highlighted. We instrumented these functions, and their counterparts on the receive end, with HPM library calls. HPM counted the average number of instructions per invocation.

Table 1 summarizes the measurements. The left panel in the table contains measurements for the high level functions of the MPICH2 port. As the table shows, blocking operations (`MPI_Send` and `MPI_Recv`) are not very good indicators of software overhead, because the instruction counts include those spent waiting for the simulated network to deliver packages. The numbers associated with non-blocking calls like `MPI_Isend` and `MPI_Irecv` are a much better measure of software overhead.

The right panel in the table contains data for message layer functions. The function `postsend` is called to post a message for sending. It includes the overhead for sending the first packet. The `senddonecb` function is called at the end of every message send. It shows the same number of instructions in every benchmark. The `recvnewcb`

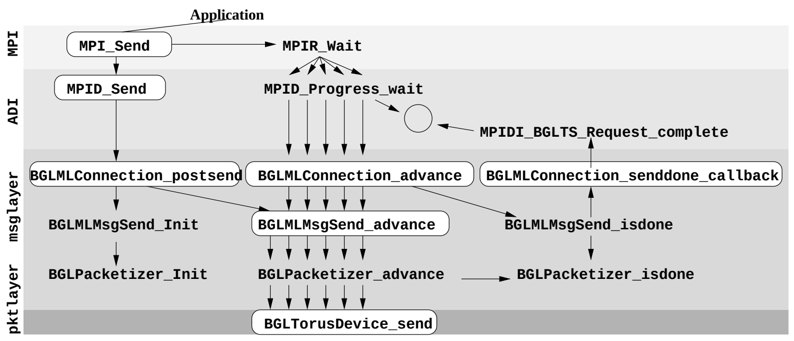

Fig. 4. The callgraph of an `MPI_Send()` call.

	FT	BT	SP	CG	MG	IS	LU
`MPI_Send`				11652	10479	3746	7129
`MPID_Send`				1759	1613	1536	1744
`MPI_Isend`		2043	2162				
`MPID_Isend`	1833	1782	1901				
`MPI_Irecv`		541	542	549	564	536	557
`MPID_Irecv`	280	279	280	293	308	280	301
`MPI_Recv`							13811
`MPID_Recv`							406

	FT	BT	SP	CG	MG	IS	LU
`postsend`	1107	1271	1401	1230	1114	1220	1265
`senddonecb`	115	115	115	115	115	115	115
`recvnewcb`	445	344	353	349	335	341	328
`recvdonecb`	16179	418	333	267	150	204	127
`advance`	2181	1643	1781	1429	1669	2865	955
`msgsend_adv`	671	653	648	620	556	642	594
`dispatch`	520	518	516	598	661	533	620

Table 1. Software overhead measurements for MPICH2 and message layer functions.

function has a slightly higher overhead because this is the function that performs the matching of an incoming message to the requests posted in the MPI request queue. The `recvdonecb` numbers show a high variance, because in certain conditions this callback copies the message buffer from the unexpected queue to the posted queue. In our measurements this happened in the FT benchmark. The table also shows the amount of instructions spent by the message layer to get a packet into the torus hardware (`msgsend_adv`) or out of the torus hardware (`dispatch`).

An `MPI_Isend` call in the BT benchmark takes about 2000 instructions. Out of these, the call to `postsend` in the message layer accounts for 1300 instructions. The `postsend` function calls `msgsend_adv` to send the first packet of the message. The `msgsend_adv` function spends an average of 650 instructions sending the packet. Thus the software overhead of `MPID_Send` can be broken down as $2000 - 1300 = 700$ instructions spent in the MPICH2 software layers, $1300 - 650 = 650$ instructions spent in administering the message layer itself, and 650 instructions spent to send each packet from the message layer.

The above reasoning points at least one place to where the message layer can be improved. The minimum number of instructions necessary to send/receive an aligned packet is 50. However, the message layer spends approximately 650 instructions for the same purpose, partially because of suboptimal implementation, alignment adjustment through memory copies and packet layer overhead. We are confident that a better implementation of the message sender/receiver can reduce the packet sending overhead by 25-50%.

7 Conclusions

In this paper we have presented a software design for a communications library for the BlueGene/L supercomputer based on the MPICH2 software package. The design concentrates on achieving very low software overheads. Concentrating on point-to-point communication, the paper presents the design decisions we have already made and the measurement methodology we are planning to use to drive our optimization work.

References

1. The MPICH and MPICH2 homepage. `http://www-unix.mcs.anl.gov/mpi/mpich`.
2. N. R. Adiga et al. An overview of the BlueGene/L supercomputer. In *SC2002 – High Performance Networking and Computing*, Baltimore, MD, November 2002.
3. G. Almasi et al. Cellular supercomputing with system-on-a-chip. In *IEEE International Solid-state Circuits Conference ISSCC*, 2001.
4. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
5. R. Brightwell and L. Shuler. Design and Implementation of MPI on Puma portals. In *In Proceedings of the Second MPI Developer's Conference*, pages 18–25, July 1996.
6. L. Ceze, K. Strauss, G. Alm´asi, P. J. Bohrer, J. R. Brunheroto, C. Caşcaval, J. G. Castanos, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and E. Schenfeld. Full circle: Simulating Linux clusters on Linux clusters. In *Proceedings of the Fourth LCI International Conference on Linux Clusters: The HPC Revolution 2003*, San Jose, CA, June 2003.
7. G. Chiola and G. Ciaccio. Gamma: a low cost network of workstations based on active messages. In *Proc. Euromicro PDP'97, London, UK, January 1997, IEEE Computer Society.*, 1997.
8. L. DeRose. The Hardware Performance Monitor Toolkit. In *Proceedings of Euro-Par*, pages 122–131, August 2001.
9. W. Gropp, E. Lusk, D. Ashton, R. Ross, R. Thakur, and B. Toonen. MPICH Abstract Device Interface Version 3.4 Reference Manual: Draft of May 20, 2003. `http://www-unix.mcs.anl.gov/mpi/mpich/adi3/adi3man.pdf`.
10. P. Mindlin, J. R. Brunheroto, L. DeRose, and J. E. Moreira. Obtaining hardware performance metrics for the BlueGene/L supercomputer. In *Proceedings of Euro-Par 2003 Conference*, Lecture Notes in Computer Science, Klagenfurt, Austria, August 2003. Springer-Verlag.
11. NAS Parallel Benchmarks. `http://www.nas.nasa.gov/Software/NPB`.
12. S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95, San Diego, CA, December 1999*, 1995.
13. L. Shuler, R. Riesen, C. Jong, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup. The PUMA operating system for massively parallel computers. In *In Proceedings of the Intel Supercomputer Users' Group. 1995 Annual North America Users' Conference*, June 1995.
14. T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, Colorado*, December 1995.
15. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.